Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects

Yuanchao Xu*, ChenCheng Ye†, Yan Solihin‡, Xipeng Shen*
*North Carolina State University
{yxu47, xshen5}@ncsu.edu
†Huazhong University of Science and Technology
yecc@hust.edu.cn
‡University of Central Florida
Yan.Solihin@ucf.edu

***Abstract*—Persistent memory has appealing properties in serving as main memory. While file access is protected by system calls, an attached persistent memory object (PMO) is one load/store away from accidental (or malicious) reads or writes, which may arise from use of just one buggy library. The recent progress in intra-process isolation could potentially protect PMO by enabling a process to partition sensitive data and code into isolated components. However, the existing intra-process isolations (e.g., Intel MPK) support isolation of only up to 16 domains, forming a major barrier for PMO protections. Although there is some recent effort trying to virtualize MPK to circumvent the limit, it suffers large overhead. This paper presents two novel architecture supports, which provide $11-52\times$ higher efficiency while offering the first known domain-based protection for PMOs.**

***Keywords*—Persistent Memory Objects, Memory Protection Keys, Intra-process Isolation**

I. Introduction

Persistent memory (PM) is emerging as a promising supplement or substitute of DRAM as main memory, offering higher density, better scaling potential, lower idle power, non-volatility, while retaining byte addressability and random access [1], [27], [30], [31]. With the right abstraction and support, PM enables data structures to be kept in memory beyond process lifetime [49]. Such Persistent Memory Object (PMO) abstraction can be *attached* (or mapped) to process address space as it uses data, and *detached* (or unmapped) from its address space afterward. Data in a PMO is long lived; its existence and structure are preserved across process runs. A PMO may be managed by the OS similar to a file (namespace and permission) but accessed like data structures (load/store instructions, pointers, etc.).

While file access is protected by system calls, an attached PMO is one load/store away from accidental (or malicious) reads or writes, which may arise from use of just one buggy library. This creates a situation where the adversary may perform memory attacks, such as *memory corruption* due to unauthorized memory writes or *memory disclosure* due to unauthorized memory reads. Countless of security attacks have been enabled by memory attacks, including code-reuse [42], [44], [47], code-injection [38] data-oriented [21] attacks, and so on. In addition to heightened *risk* of data disclosure or corruption, PMO also suffers from heightened *cost* of such memory attacks, as it keeps valuable data that is long lived.

In this paper, we try to answer the question of *how to improve the security of a PMO that is attached to a process*. We point out that memory attacks on PMO data may arise *spatially* (e.g., when a thread of a process that is not authorized accesses the PMO attached in this process) or *temporally* (e.g., when a thread accesses the PMO beyond its authorization window). The goal of this work is to provide a process spatio-temporal protection of PMOs.

Intra-process isolation techniques enable a process to partition sensitive data and code into isolated components. By specifying access policy via applying the principle of least privilege to each isolated component (e.g. a group of pages), intra-process isolation limits the influence of bugs and vulnerabilities to one component. One such intra-process isolation support is Intel Memory Protection Key (MPK) [22], which extends the x86 Instruction Set Architecture (ISA) with the capability of defining *domains*. With MPK, a process address space can be partitioned into up to 16 domains, with each domain represented by a protection key. A new 32-bit register, PKRU, is added to each logical core, providing a way to express the access policy (read/write) of each domain for each thread. The PKRU is integrated with the TLB checking mechanism to allow the memory management unit (MMU) to enforce the policy.

Intra-process protection may be used to provide spatio-temporal protection for PMOs. The basic idea is that when a process attaches a PMO, the PMO is placed into a protection domain, and access control policy is set for the domain. When a thread accesses PMO, the corresponding load or store is checked against the access policy of the domain for the thread, as well as the page access policy from the TLB or page table. The more restrictive permission is derived to determine the legality of the access.

However, using MPK to support intra-process isolation of

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00062

PMOs runs into security and scalability challenges. First, MPK supports only 16 protection keys, which is too few. After 16 keys are allocated, further pkey_alloc() call will return with an error, which forces the programmer to either forgo the use of domains, or reuse an old domain for multiple unrelated PMOs. For example, consider a typical server application, which spawns a thread to interact with a client in response to a connection request by the client. The thread may store user-private data (persistently in a PMO in our case). The Heartbleed vulnerability targeting OpenSSL demonstrates that a vulnerable library allows the attacker to steal sensitive data such as private keys and passwords [41]. Allocating different users' data in separate domains improves security by isolating each user data from other threads that are not meant to access it. Having too few keys forces data from multiple clients to share a single domain and key, allowing a compromised thread to access data intended for other threads. Therefore, the number of domains should ideally be high. As a starting point, in Linux a process can open 1024 files simultaneously, some server applications may allow thousands of connections; so at least several thousands of simultaneously attached PMOs should be supported. Extending MPK to support several thousand domains is not feasible as it requires extending the PKRU register to several kilobytes in size, which requires substantial changes in the ISA and may affect critical path delays in the pipeline.

A recent effort, *libmpk* [39], circumvents MPK's limit of 16 domains via software-based virtualization. libmpk supports a large number of domains but map only 16 of them to protection keys. If the program accesses only mapped domains, no performance overhead results. However, if it accesses an unmapped domain, an exception is triggered, and the exception handler selects a domain to unmap and reassigns the key to the new domain. This step involves very substantial overheads including rewriting the domain field in affected page table entries of the victim and new domains, TLB shootdowns of all cores, writing to PKRU, etc. As a result, libmpk suffers from a large runtime overhead ($17.4\times$ slowdown for each permission update on 35 domains [39]).

To support a large number of protection domains efficiently, we propose two novel architecture mechanisms. The first design, *Hardware MPK Virtualization*, builds on MPK while giving an illusion of unlimited domains. Similar to libmpk [39], at any given time, only 16 domains map to keys. When an unmapped domain is accessed, a victim domain is selected and unmapped, and its key is reassigned to the new domain. Different from libmpk, we provide MMU-like support for handling domains, including a radix-tree Domain Translation Table that can be walked by a hardware handler, and Domain Translation Lookaside Buffer (DTLB) that caches the table for fast access. Much of the remaining architecture is unmodified from MPK.

The second design, *Hardware Domain Virtualization*, is more aggressive; it removes the need for limited keys altogether. It manages a large number of domains without mapping them to keys first. It manages per-thread access control directly on domains, offering even greater flexibility in domain-based protections. A key benefit to this design is removing the need for TLB shootdown when an entry is evicted or changed.

Overall, this paper makes following major contributions:

1) We propose to improve the security of PMOs from memory attacks by assigning each attached PMO to a protection domain, providing intra-process isolation of PMOs.
2) We propose an architecture support for efficient MPK virtualization, which supports a large number of domains sharing a limited number of protection keys. This is built on top of MPK.
3) We propose an architecture support for domain virtualization, which manages per-thread permission directly on domains, completely removing the mapping of domains to a limited number of keys.
4) We evaluate both schemes and show that they perform $11\times$ and $52\times$ speedups over libmpk, the state-of-the-art software MPK virtualization.

II. Background

This work focuses on intra-process isolation, that is, the isolation of accesses among different threads that share the same address space. Inter-process isolation is relatively easier as different processes have different address spaces, and explicit APIs with permission control flags can be used when sharing a PMO among processes (similar to the APIs for shared memory management).

A. Intra-Process Isolation Support

Intra-process isolation has been long recognized as important for security, especially as program complexity is increasing. One approach to such isolation is capability-based addressing [56], [57], where pointers are replaced by protected objects (called capabilities), which specify the objects that the pointers can legally refer to. A recent example is CHERI [56], [57], where a fat pointer specifies legal bounds and permissions, and a co-processor check for access validity. CODOM [51] needs dramatic hardware changes to achieve efficient intra-process isolation. A less comprehensive (but simpler) solution is to add explicit code to check bounds of pointer references [43], [43], [53]. Hardware support for accelerating bounds check, such as Intel MPX, has been proposed, and shown to incur a much lower slowdown (e.g. about 30% [29]).

A complementary approach to intra-process isolation is to focus on the objects being referenced, rather than the pointers that de-reference them. It specifies, for a given memory object (such as pages), when (temporal) which threads (spatial) can access the pages. Example hardware page protections [7], [8], [12], [32], [35] support memory isolation and provide near zero overhead within a component. But switching between components still involves a switch to kernel mode, which incurs substantial overhead. Intel MPK [22] is an example of hardware support implementing this approach, explained next.

B. Intel Memory Protection Key

The recently released Intel Memory Protection Key (MPK) extension [22] allows the address space to be partitioned into 16 disjoint *domains*, and each domain is represented by a protection key. A new 32-bit register, PKRU, is added to each logical core to specify and enforce thread-specific permission (read and/or write) for each key. Each key uses one bit for read permission and one bit for write permission. Two non-privilege user-level instructions are provided to write and read the PKRU register: WRPKRU instruction writes a new value to PKRU register (˜11-260 cycle latency), while RDPRKU instruction reads the current value of PKRU. Three system calls are implemented in Linux kernel: `pkey_alloc()` allocates an unused key from 16-bit bitmap in kernel, `pkey_free()` frees a key and marks it available, while `pkey_mprotection` associates a key with page table entries (PTEs) by changing the key value in all PTEs.

MPK can be used to support executable-only memory by changing the domain permission as inaccessible in the PKRU register. Code can still jump to this domain and execute code but all reads and writes are prohibited for this domain.

After associating a domain with a protection key, writing it in PTEs, and setting the appropriate PKRU value for the domain, a memory access to this domain will result in the PTE (along with the protection key value) cached in a TLB entry. The access will read the protection key value from the entry, which then indexes the PKRU to obtain the permission bits associated with the key. The access is legal if the both domain permission (in PKRU) and page permission (in TLB) allow the access.

C. Persistent Memory Programming Support

Non-volatile memories (NVMs), such as Intel Optane DC Persistent Memory, provides high capacity at low cost, low idle power, byte-addressability, persistence, and performance similar to DRAM [1], [27], [30], [31]. There are at least two paradigms for using NVM. One uses it as storage to host a file system, the other uses it via a new abstraction where a data structure is wrapped into a *persistent memory object* (PMO), which allows the data structure to be hosted persistently in physical memory without involving a file system. PMOs may combine some features of a file system (naming, permission, durability, and sharing) and some features of data structures (pointer-rich, address space mapping, purely load/store access). In this paper, we assume the latter.

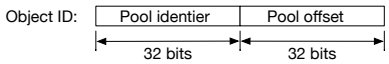

Fig. 1. Structure of pool pointer [11], [54], [55]

A PMO may be a container for a data structure that lives beyond process termination and system reboots. A PMO requires several features to be supported: *crash consistency* allowing a PMO to remain in a consistent state even on process crashes or system power loss, OS-managed *namespace* and *permission* allowing the PMO to be found on recovery, *attach* and *detach* primitives allowing the PMO to be attached to a process address space when needed and detached afterward, *relocatability* allowing the PMO to be attached at virtual address different from the one from the previous session [60]. PMOs may be implemented as pools [11], [14], [23], [54], [55], each given a unique identifier. A pool may be organized as a collection, with a root object from which all other objects in the pool can be reached. In this paper, we use the term PMO as a general concept, and pool as a specific implementation of a PMO, which may not have all features a PMO should support.

To support relocatability, each pointer (64-bit) used in a data structure is split into a 32-bit pool ID (ObjetID) concatenated with a 32-bit offset within the pool (Figure 1). To address a pointer, the base address for the ObjectID is added to the offset. PMDK [23] and other prior works [14], [54] have described interfaces for manipulating pools and objects. We adopt the interface proposed by Wang *et al.* [54]. It supports functions for creating pools or objects within pools, supports mechanisms for persisting objects and failure-safety through durable transactions. Table I shows a subset of their interface. Our design is compatible with software [11], [14], [23] and hardware [54], [55] support for PMO relocatability.

TABLE I
POOL APIS DESCRIBED IN PRIOR WORK [23], [54].

Function	Description
pool* pool_create (name, size, mode)	Create a pool with the specified size and associate it with a name. The running process is the owner.
pool* pool_open (name, mode)	Reopen a pool using name that was previously created. Permissions will be checked.
pool_close(pool* p)	Close a pool p
OID pool_root(pool* p, size)	Return the root object of the pool p with specific size. The root object is intended for programmers to design as a directory of the contents in the pool.
OID pmalloc (pool* p, size)	Allocate a chunk of persistent data with the given size on pool p and return the ObjectID of the first byte.
pfree(oid)	Free persistent data pointed to by the ObjectID.
void* oid_direct(oid)	Translate an ObjectID to a virtual address. Used when there is no hardware translation.

III. THREAT MODEL

Just like any other data structures, data structures in PMO may contain buffers and pointers. Code that accesses PMO may contain regular known vulnerabilities. Our mechanism seeks to make unauthorized reads or writes to data in PMOs difficult by applying the principle of least privileges; unauthorized reads or writes are the fundamental schemes many types of memory attacks rely on.

The attacker may compromise a thread of the same process and try to exploit the memory vulnerabilities. While a PMO is attached, the attacker may attempt to read or write data in the PMO. We do not assume that the attacker has an ability to arbitrarily inject or execute arbitrary code (if he/she could, there are not many protection schemes that are effective against it).

We assume trusted system software, such as the OS, which manages address space isolation between processes, and compiler, which generates code correctly given user program. Furthermore, we assume that trusted hardware, like processor memory management unit (MMU), is implemented correctly,

User-level permission change instructions can only be inserted by the programmer or compiler. We can prevent the attacker from injecting or reusing these instructions (e.g. through ROP) by implementing call gates and performing binary inspection and rewriting similar to ERIM [50].

Side-channel and rowhammer attacks, and microachitectural leaks, although important, are beyond the scope of this work.

IV. Design

A. Protection Goals

In order to protect PMO data from accidental or malicious reads/writes, we apply the principle of least privilege by granting PMO access permission only to the threads that need to access it (spatial isolation), and only when they need to access it (temporal isolation). In particular, we require that an access to PMO from a thread is legal only if (1) the page has the appropriate read/write permission, (2) the process has attached the PMO, and (3) the thread has read/write permission to the domain associated with the PMO.

To achieve the first requirement, we rely on traditional virtual memory mechanisms for enforcing per-page process-specific permission.

To achieve the second requirement, we assume that PMOs are managed by the OS and laid out in physical memory, either contiguously or non-contiguously (with embedded page table support). To attach a PMO to its address space, a process makes a system call specifying the PMO path/name and the requested permission. If a PMO is successfully attached, the system call returns a PMO ID which is also the domain ID. A PMO can map only to an aligned and contiguous range of virtual address that corresponds to the granularity of the hierarchy level of the page table. For example, the smallest PMO occupies 4KB VA region, the next larger PMO occupies 2MB VA region, and then 1GB, etc., corresponding to the level in the page table. Note that the PMO does not have to use the entire VA range allocated to it.

A process can express intent to read (R) or both read and write (RW) to the PMO. The system call ensures that the OS can grant attachment requests only if the user who owns the process is allowed to attach the PMO. The system call will also enforce inter-process isolation, locking, and sharing policy. For example, a PMO may be attached exclusively to only one process for writing, but may be attached to multiple processes for reading. The system may also keep a finer grain permission scheme based on *attach key*, where a process must produce the correct key for the attach request to be granted. This allows an additional restriction to specify which user processes should be allowed to attach a PMO. The system may also detach a PMO from a process upon request or automatically when appropriate, for example when a process terminates prior to detaching a PMO, or when it is suspected to have been compromised by security attacks.

To achieve the third requirement, we start by an observation that protection domains, such as Intel MPK, is a good fit for PMOs for its spatio-temporal protection capability. The spatial protection allows permission for a domain/PMO to be defined differently for different threads, providing inter-thread protection where only a thread with sufficient permission can access the domain. The temporal protection allows permission to be added or removed for the same thread over time, providing intra-thread protection. To add or remove permission, we introduce a *user-level instruction* `SETPERM`. `SETPERM` takes a domain ID as a source operand, and a read/write flag as the second source operand. The instruction allows granting access to a PMO by setting (or unsetting) read or read/write permission for the thread that executes it. `SETPERM` is similar to `WRPKRU` MPK, but with a difference that it only sets permission for one domain, whereas `WRPKRU` simultaneously sets the permission for 16 protection keys using all 32 bits. Therefore, `SETPERM` works with a large number of domains, unlike `WRPKRU` which is limited to 16 keys.

The `SETPERM` instruction must be compatible with the processor memory consistency model. With sequential consistency, processor consistency, and total store ordering (TSO), it is treated as a store instruction. For more relaxed consistency models, such as weak ordering or release consistency [48], it is treated as a full memory fence/barrier. As such, we are guaranteed that any loads/stores older than it are performed prior to allowing any younger loads/stores to perform. The appropriate PMO domain permission for this thread is changed at the fence point.

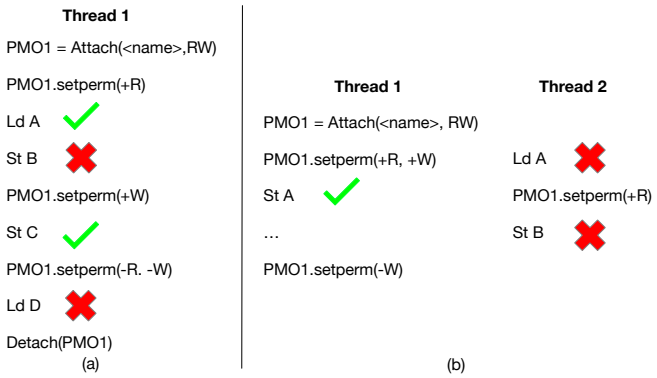

Fig. 2. Example of: (a) Intra-thread (temporal) protection (b) Inter-thread (spatial) protection.

The domain-based PMO isolation is illustrated with an example in Figure 2. Assume that addresses A, B, C, and D

reside in PMO1. Part (a) illustrates temporal isolation. First, Thread1 attaches PMO1 with intended read/write permission to the process address space. This does not yet grant any threads to read/write to the PMO/domain, until it sets the per-thread read permission (+R). The subsequent ld A is permitted but st B is denied. After adding write permission (+W), st C is permitted. When read and write permissions are removed (-R, -W), ld D is denied. The permission setting is thread specific, as illustrated in part (b) of the figure. For Thread1, st A is permitted. For Thread2, ld A is denied because Thread2 has not obtained permission, and st B is also denied because the permission is insufficient. The instructions to add/remove domain permission for a PMO must be inserted by the programmer (e.g., through API or #pragma), or by the compiler based on program analysis.

B. Number of Protection Domains

We draw a distinction between domains and protection keys. Each attached PMO is assigned a domain. But current architectures manage permission based on protection keys; hence there is a gap between our goal and the current architecture support. More importantly, Intel MPK supports only 16 protection keys, which may result in compromised security if the programmer forgoes the use of domains or reuses a domain for multiple unrelated PMOs.

Consider an example of a typical server application, which spawns a thread for each connection request by the client. The thread may store user-private data (persistently in a PMO in our case). The Heartbleed vulnerability targeting OpenSSL demonstrates that a vulnerable library allows the attacker to steal sensitive data such as private keys and passwords [41], so allocating different users' data in separate domains improves security by isolating each user data from other threads that are not meant to access it. Having too few keys forces data from multiple clients to share a single domain and key, which may reduce security. For example, suppose that thread 1 should have read permission for PMO A but read/write for PMO B, denoted as $R_1(A)$ and $RW_1(B)$, respectively. If A and B share one protection key X, then the permission of X must be the least restrictive of A and B, which is RW. However, setting $RW_1(X)$ means that the thread can write to A even though it should not. Hence, the security protection has weakened. Furthermore, incompatibility between threads complicates this grouping. Suppose that $RW_1(B)$ and $RW_1(C)$ but $RW_2(B)$ and $None_2(C)$, forcing B and C to share a key does not weaken permission for thread 1 but weakens for thread 2. Despite the best clustering analysis to group domains with similar permissions across all threads, we will still have cases where security is weakened due to the limited number of protection keys. How large the appropriate number of supported domains should be is an open question. A large number of domains can provide protection flexibility, especially for server applications which may spawn many threads with each serving a different user. As a starting point, in Linux a process can open 1024 files simultaneously, some server applications may allow thousands of connections; so at least several thousands of simultaneously attached PMOs should be supported.

Extending MPK to support several thousand domains is not feasible as it requires extending the PKRU register to several kilobytes in size, which is not feasible for several reasons. First, PKRU is read into or written from a general purpose register, so its width must match the width of such a register (e.g. 32 bits for EAX/EBX/ECX/EDX). Second, its checking time must fit within the number of clock cycles allocated for TLB check, hence if too large, the critical path delay of TLB checking will be affected.

C. High-Level Design

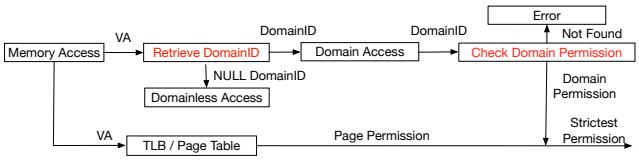

Fig. 3. Illustrating how domain protection is integrated into the MMU.

At a high level, domain-based protection is integrated into the MMU as steps performed in parallel with traditional page-based permission checking, as illustrated in Figure 3. When a load/store virtual address is available, it is used to retrieve domain ID and the domain ID is used to check the domain permission for the thread. In parallel, the TLB or page table is checked for traditional page permission. The two permissions are compared to derive the strictest permission, which determines the legality of the access. The parallel checking avoids adding to the critical path of access. Furthermore, not all applications may need domain protection, hence a NULL domain ID is reserved to indicate that domain checking is unnecessary. Next we will discuss the proposed techniques: MPK virtualization and domain virtualization.

D. Hardware-Based MPK Virtualization

Figure 4 illustrates the design and mechanism of our hardware MPK Virtualization. Recall that in this design, we build on top of MPK, preserving most of its features and structure. Thus, we must add a mechanism to allow mapping a large number of domains to 16 protection keys. This is accomplished by keeping the mapping of domains to keys using a Domain Translation Table (DTT). DTT is an OS-managed data structure created for each process that uses domain protection. It is indexed by virtual address (VA) and each entry contains the domain ID, current protection key the domain ID maps to, and permission for the domain. Since the address space of a process may be sparse, the DTT is organized hierarchically, similar to a page table. In the figure, DTT is shown to have a two hierarchy level, because the example PMO occupies a 2MB region. Analogous to the TLB as a cache for the page table, Domain Translation Table Lookaside Buffer (DTTLB) is the cache for the DTT to allow fast mapping of VA to protection key. However, whereas DTT

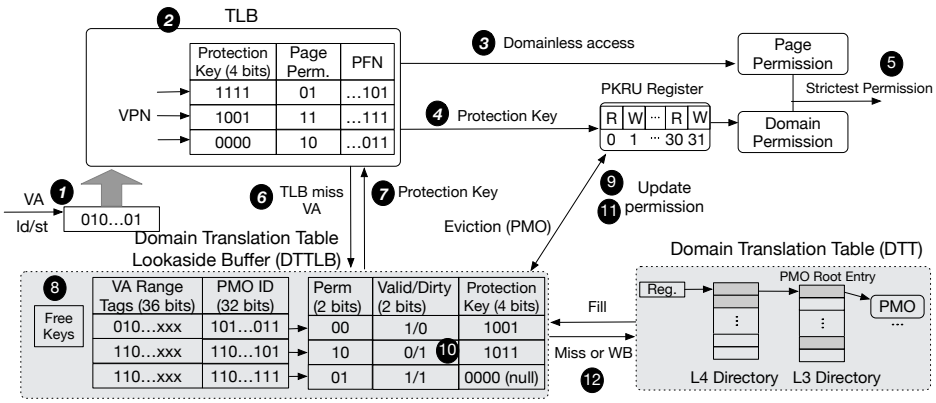

Fig. 4. Diagram of the MPK virtualization scheme.

keeps permission for all threads in a process, DTTLB only caches the permission for the thread that currently runs in the core.

A DTTLB/DTT entry contains a 36-bit VA range tag, a 32-bit PMO/domain ID, a Valid bit, a Dirty bit, and a 4-bit protection key. The VA Range allows each entry to represent an entire domain expressed as its base VA and domain size. Alternatively, since a PMO occupies contiguous and aligned VA range that corresponds to page sizes (4KB, 2MB, or 1GB), the VA range can simply be the base address with a two-bit field to indicate which size it uses. The 4-bit protection key represents which key a PMO/domain ID currently maps to. A NULL key value (0000) is reserved to indicate that this PMO is domainless. The "Free Keys" structure keeps all keys that are not mapped. There are two kinds of entries in the DTT, *directory entry* and *PMO root entry*. One *valid bit* and one *next level bit* are introduced in both kinds of entries. *Next level bit* indicates the next level is either a directory (1) or a PMO (0). A dictionary entry points to the physical frame number (PFN) of the next level directory. The root entry of a PMO stores its ID. DTT is pointed by a register for looking up.

This design introduces no changes to the TLB, page table structure, or MPK mechanism. DTTLB can be quite small, even 16 entries are sufficient to hold all 16 domains that map to protection keys, making it feasible to use content-addressable memory (CAM) for associative lookup. However, DTTLB can have more than 16 entries to hold information of all domains in the thread working set.

a) Handling a TLB Hit: A TLB hit is handled identically to the MPK mechanism. When a ld/st accesses a VA (❶ in Figure 4), the TLB is checked for a match ❷. On a match, a protection key is read out. If the key is NULL (0000), this access is domainless ❸. If the key is not NULL, the key is used to index the PKRU register to obtain its domain permission ❹. If the load/store is legal according to both the domain permission and page permission ❺, the load/store is allowed to access the cache. Otherwise, an exception is raised.

b) Handling a TLB Miss: A TLB miss is handled differently from MPK. On a TLB miss, the VA is checked against VA range tags in the DTTLB ❻. If a match with a valid entry is found, the protection key is read and supplied to the TLB to be combined with other information obtained from page table walk ❼. A match with an invalid entry indicates that the domain is not currently mapped to a key. If a free key is available, the key is then assigned to the domain ❽, PKRU is updated to reflect it ❾, and the valid bit is set. If a free key is not found, a victim domain is selected, based on a replacement policy (Pseudo LRU in our implementation). Then, the key is reassigned from the victim domain to the new domain and the DTTLB entry of the victim domain is marked invalid and dirty ❿. The DTTLB entry of the new domain is marked valid and dirty. Then, the PKRU is updated to reflect the permission of the new domain associated with the protection key ⓫. TLB shootdown is then initiated (Range_Flush of the victim PMO VA range) for all cores in order to invalidate the victim pages' mapping to the protection key. The accessed page in the new PMO fills the TLB with the protection key mapping. If a DTTLB miss occurs, the DTT is walked to find both its domain information (ID and permission). DTTLB updates the DTT lazily; when a dirty DTTLB is evicted, its protection key mapping updates the DTT ⓬.

The entries in DTT are added/removed by the attach and detach system calls. The instruction `SETPERM` updates the permission information in a DTT entry, and will result in invalidating the corresponding entry (if cached) at the DTTLB.

c) Security Assessment: Let us now discuss the impact of MPK virtualization on PMO security protection. First, we note that spatio-temporal domain protection requires that (1) Every memory access must be checked to identify its domain, and (2) For an access to a certain domain, its legality is checked against the domain access permission for the thread.

The proposed method meets both requirements. When a PMO is attached, the PMO/domain ID and its VA range are added as a new entry in the DTT. All memory accesses to this VA Range that suffers a TLB miss will check the DTTLB (if hit) or trigger a DTT walk (if miss). Domain accesses to PMOs find its PMO ID in TLB, DTTLB, or DTT, and they are treated as domain accesses. An access that does not find a domain

in the DTT is a domainless access and recorded with NULL domain in the TLB. When domain-to-key mapping changes in the DTTLB, TLB shootdown ensures that TLB entries are invalidated if their mapping is affected, while the PKRU is updated to reflect the permission of the new domain. Hence, both requirements are met.

Care must be taken on context switch to continue meeting the requirements. Because PKRU and DTTLB entries are thread specific, it must be flushed upon a context switch. Any dirty entries in the DTTLB must be written back to the DTT prior to the switch. In MPK, the PKRU is part of the process state that is saved and restored. In our design, because DTT contains information of all domains and permission of that domain for all threads, hence the content of DTTLB and PKRU can be reconstructed when the thread resumes in the future, hence they can be flushed.

d) Comparison with libmpk: Our design efficiently and transparently supports a large number of domains with minor hardware modifications by virtualizing the assignments from PMO IDs to protection keys. When an access to an unmapped domain occurs, libmpk incurs an exception that triggers an exception handler to unmap and map the domain by writing to as many PTEs as the affected domain has. In contrast, DTTLB allows the unmap and map to occur in hardware and changes are reflected in the PKRU. Both libmpk and our MPK virtualization involve TLB shootdowns, however, the cost of shootdowns is proportional to the size of TLB, while libmpk's PTE changes is proportional to the domain size. Hence, our MPK virtualization is both faster and more scalable. Furthermore, with our solution, programmers do not need to memorize the assignment from PMO IDs to protection keys, or explicitly handle the assignment and reassignment from PMO IDs to protection keys in the PKRU register. They can simply change the permission of a PMO ID.

E. Hardware-Based Domain Virtualization

The first design, hardware MPK virtualization, supports a large number of PMOs/domains while leveraging existing MPK hardware as much as possible. However, a critical drawback is that everytime a domain-key mapping changes, TLB shootdown must be initiated to invalidate stale VA-to-key information in TLB entries. As the number of domains sharing 16 protection keys increases, domain-key remapping becomes more frequent, triggering frequent TLB shootdowns. Thus, we need a more scalable alternative design. We propose *hardware domain virtualization* that obviates the need for TLB shootdowns.

a) Architectural Design: This design foregoes MPK and introduces a new mechanism to enable direct permission lookup, as illustrated in Figure 5. It makes a minor change to the TLB by adding a 10-bit domain ID in each TLB entry in place of the protection key ID. If there is a TLB miss, the domain ID for a page is retrieved from the Domain Range Table (DRT), an OS-managed data structure. DRT is organized similarly to DTT with a hierarchical table, but without keeping domain permission information. DRT may have directory entry or PMO root entry. Each entry of either type has a *valid* bit and a *next level* bit. The *next level* bit indicates whether the next level is a directory (1) or a PMO (0). The directory entry has a 36-bit page frame number (PFN) of the next level dictionary.

The permission information for domains and threads is kept using a separate table called the Permission Table (PT), another OS-managed data structure. It is indexed by domain ID and thread ID, and contains the domain permission for the thread. To provide fast permission check, this information is cached in a hardware structure called Permission Table Lookaside Buffer (PTLB). A PTLB entry contains a 10-bit domain ID used as tag, a 2-bit permission, and a dirty bit. The permission can be $1x$ (inaccessible, execute only), 01 (read-only), or 00 (readable and writable).

b) Operations: DRT and PT entries are added or removed in reaction to the attach or detach system calls. As before, each PMO is assigned a unique domain ID. PTLB miss results in retrieving the domain permission information for the thread from the PT. Permission change requests to a domain (SETPERM) can be completed entirely in the PTLB by directly changing the domain permission. The dirty bit for such an entry is set when its domain permission changes. When a dirty PTLB entry is evicted, the permission is written back to the PT.

To check permission for a load/store instruction ❶, first the VA is used to access the TLB ❷. If we have a TLB hit, the domain ID and page permission are retrieved ❸. If the domain ID is NULL, the access is a domainless access and no further action is taken. Otherwise, the retrieved domain ID is used to look up the PTLB to retrieve the domain permission ❹. If the load/store is legal according to both the domain permission and the page permission ❺, the load/store proceeds. Otherwise, an exception is raised.

If we have a TLB miss ❻, page table walk *and* DRT walk are performed in parallel. The physical address obtained from page table walk and domain ID obtained from DRT walk are combined into the new TLB entry ❼. If the VA is not found in the DRT after the walk, it does not belong to any domain, so a NULL domain is used. After the TLB, PTLB is checked. If we have PTLB hit, domain permission is retrieved and used. If we have a PTLB miss ❽, a victim PTLB entry is selected to make room for a new entry, and the PT is looked up to retrieve the domain permission for the new entry ❾.

c) Security Assessment: As with MPK virtualization, domain ID for an access is always retrieved (via TLB and DRT), and domain permission for the thread is always checked (via PTLB and PT). Both security requirements are hence met.

Handling context switches requires flushing thread-specific information in the PTLB, but not the TLB. Any dirty entries in the PTLB are first written back to the PT, then all entries can be flushed. The information of domain ID in the TLB remains valid. As the PT has only a few entries (16 in our base case), the impact of flushing it on context switch on performance is small.

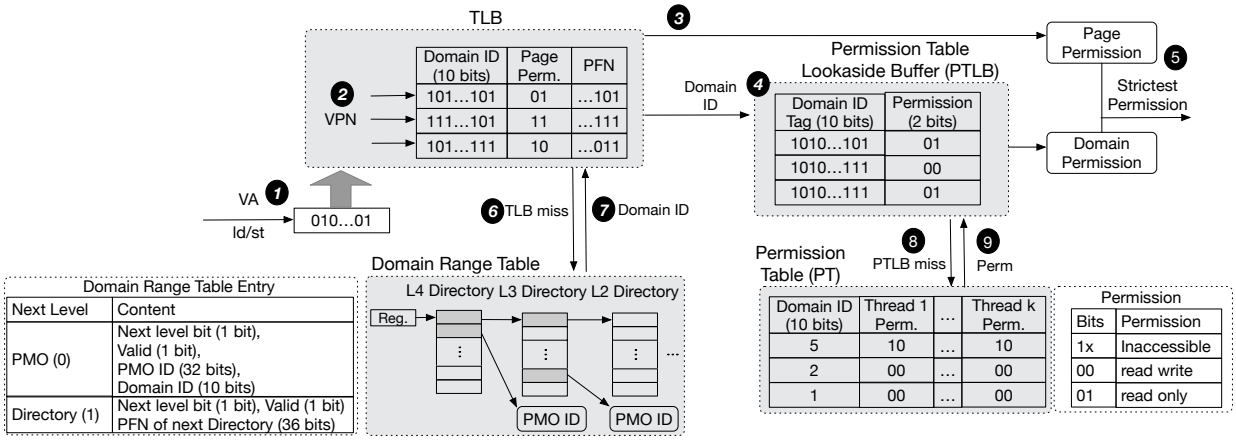

Fig. 5. The Domain Virtualization Design and Mechanism.

V. Evaluation Methodology

We base our simulator on Sniper simulator [10], a cycle-accurate X86 simulator. PMOs are implemented as memory mapped regions. We evaluate the following schemes. The first scheme is non-protected execution serving as the *baseline*. Intel Pin [36] was used on a real machine to obtain a trace that is then fed to the simulator to obtain baseline performance. The second scheme is an ideal MPK virtualization (*lowerbound*), which represents a case where no overhead is added to MPK except for programming of PKRU through WRPKRU instructions. We insert WRPKRU instructions to enable each PMO access and disable it afterward. We execute the program with Pin to obtain its trace, which is fed into the simulator to obtain its performance. One can think of this scheme as having MPK virtualization without any penalties for accessing the DTTLB or DTT. The third scheme we evaluate is the realistic version of the proposed scheme, obtained by feeding into Sniper the trace with WRPKRU instruction and the architecture overheads introduced in our schemes. The parameters used in our schemes are shown in Table II. Sniper does not support the WRPKRU instruction and regards it as unknown, hence we add appropriate delays (27 cycles) to executing the WRPKRU instruction when we re-execute the trace in the simulator.

Our experiments of architectural overhead are based on the design logic in Section IV and overhead values in Table II. In the setting, the DTTLB/DTT table walk can be executed in parallel with traditional page table walk. DTT table walk latency is the same as or smaller than page table walk latency. On a DTTLB hit, the latency of DTTLB add/modify/search is always smaller than page table walk latency of TLB misses. So there is no extra overhead on TLB miss. TLB invalidation overhead is the sum of the overhead for a key remapping for $number\_of\_thread$ threads. The subsequent TLB misses resulting from TLB invalidations is also taken into account. For the domain virtualization design, there is no extra overhead on a TLB miss since the DRT and page table can be walked in parallel and the DRT is shallower than the page table.

TABLE II
Simulation parameters.

Processor	2.2 GHz, 4-way issue Out-of-order, 128-entry ROB, Intel x86-64 architecture, Pentium M branch predictor
Cache	L1D cache 8-ways 32KB, 1 cycle access time; L2 cache: 16-ways 1MB, 8 cycles access time
Memory	DRAM latency: 120 cycles; NVM latency: 360 cycles; 64 GB/s Bandwidth; Directory-based MESI protocol
TLB	L1 data TLB: 4KB pages, 4-way, 64 entries; L2 4KB/2MB pages, 6-way, 1536 entries; 1 cycle L1 TLB access, 4 cycles L2 TLB access; 30 cycles TLB miss penalty
MPK	WRPKRU: 27 cycles
MPK Virtualization	DTTLB: 16 entries; Free keys check/update: 1 cycle; DTTLB hit: 1 cycle; Add/Remove/Modify DTTLB entry: 1 cycle; DTTLB miss: 30 cycles, PKRU update 1 cycle; TLB invalidation: 286 cycles
Domain Virtulization	PTLB: 16 entries; PTLB access: 1 cycle; PTLB miss (incl. permission table lookup): 30 cycles; Add/Remove/Modify PTLB entry: 1 cycle

We assume main memory consists of DRAM and NVM. The NVM latency is $3\times$ higher than DRAM latency, in line with Intel Optane DC Persistent Memory characterization [24]. PMO accesses use NVM latency while other accesses use DRAM latency.

TABLE III
WHISPER Benchmarks [37] and their configurations.

Benchmark	Description
Echo	echo test, 100k transactions in total
YCSB	YCSB like test, 80% writes, 100k transactions in total
TPCC	TPC-C like test, 80% writes, 100k transactions in total
C-tree	100K insert operations
Hashmap	100K insert operations
Redis	redis server/ lru-test, 1 million gets/puts

Single PMO on WHISPER Benchmarks: WHIPSER

benchmarks are based on real world persistent memory (PM) applications, including PM key-value stores Echo and Redis, a PM database N-store, and PM transactional libraries. Although each of its benchmarks uses only a single PMO, evaluations on them can help measure the inherent overhead from applying domain protection on real world persistent memory applications and how our schemes affect the overhead [37].

As listed in Table III, we execute WHISPER benchmarks using 100k transactions or operations in a 2GB PMO of single thread. We assign the entire PMO with a protection key through `pkey_alloc()` and `pkey_mprotect()`. The default permission for this key is inaccessible. We insert `pkey_set/WRPKRU` before and after every PMO access to enable or disable the access to this protection key associated with the PMO.

TABLE IV
MICROBENCHMARK DESCRIPTION.

Benchmark	Description
AVL Tree (AVL)	Insert or delete nodes in the tree.
RB tree (RBT)	Insert or delete nodes in the tree.
B+ tree (BT)	Insert or delete nodes in the tree.
Linked List (LL)	Insert or delete nodes in the linked list.
String Swap (SS)	Randomly swap strings in the string array.

Multi-PMO on Micro Benchmarks: To study the impact of multiple PMOs, we leverage the benchmarks used in prior NVM studies [11], [14], [26], [34], [46], as shown in Table IV. Each benchmark has 1024 consecutive PMOs, and each of them is 8MB in size. Each PMO is a pool of nodes for the data structures in Table IV. The main data structures contain nodes in different PMOs with each node containing a 64-byte value except B+Tree, in which a node is 4096-byte long, containing 126 values and two pointers. Every operation randomly selects a node in a PMO to operate on. To experiment with different numbers of active PMOs, we vary the set of PMOs such that the largest number of PMOs ranges from 16 to 1024 with a 16 stride. Compared to the WHISPER experiment, we enable the write permissions of a PMO before and after every data structure operation rather than on every PMO access instruction. The application has read permission for all PMOs. Every data structure starts with 1K initial nodes. Each benchmark executes 1 million operations on the data structure, in which, 90% instructions are insert operations.

VI. EVALUATION

This section first reports the overhead of the two proposed solutions with a single PMO on WHIPER, compared to the execution of the default MPK. It then reports the performance on multi-PMO benchmarks, compared to the previously proposed software-based MPK virtualization, *libmpk* [39]. It finally provides the area space overhead and security analysis of the solutions.

A. Single-PMO Results on WHISPER

Table V reports the overhead on WHISPER; the baseline is the performance of the default runs without protection. The second column in the table reports the rate of permission switches, calculated as number of switches per second. (The permission to the PMO is granted before each PMO access and disabled after that access). The overheads from MPK on these benchmarks range from 0.77% to 2.65%. Our first design, hardware MPK virtualization, enjoys the same performance as the default MPK because the benchmarks have only one PMO hence do not need to evict any protection keys. Our hardware domain virtualization shows slightly higher overheads, 0.85–2.91%, because PTLB permission lookup increases the latency of each PMO access even though the data may be in the cache. The overhead on TPCC is the largest due to a higher percentage of PMO accesses in the program.

TABLE V
OVERHEAD OF MPK VS. HARDWARE MPK VIRTUALIZATION AND DOMAIN VIRTUALIZATION FOR WHISPER WITH A SINGLE PMO.

Benchmarks	Switches/sec	Overhead (%)		
		Default MPK	Virtulization	
			MPK	Domain
Echo	712,631	0.77	0.77	0.85
YCSB	1,152,379	1.48	1.48	1.63
TPCC	951,529	2.65	2.65	2.91
C-tree	839,138	1.21	1.21	1.30
Hashmap	863,251	1.05	1.05	1.14
Redis	1,038,506	1.28	1.28	1.41
Average	926,239	1.41	1.41	1.54

B. Multi-PMO Results

This section presents the performance measurements on the multi-PMO benchmarks (16 to 1024 PMOs per program). The access permission to a PMO is granted before a data structure operation modifies it and disabled right after the completion of the operation to minimize the security vulnerability. Table VI reports the frequency of the permission switches. The "lowerbound overheads" column in the table reports the overheads from just executing write permission-granting and disabling instructions, providing the lowerbound of the security protection overheads. The lowerbound overhead is related to the number of switches per second.

TABLE VI
LOWERBOUND OVERHEAD AND PERMISSION SWITCH FREQUENCIES FOR THE MULTI-PMO BENCHMARKS

Benchmark	Switches/sec	Lowerbound overheads
AVL Tree (AVL)	2,326,578	3.28%
RB tree (RBT)	1,594,634	2.25%
B+ tree (BT)	2,085,772	2.94%
Linked List (LL)	305,388	0.43%
String Swap (SS)	3,636,006	5.12%

Figure 6 shows how overheads of various schemes compare, when the number of PMOs varies over the x-axes. The y-axes shows the execution time overhead percentage over lowerbound, e.g. 2^2 means 4% slower, 2^4 means 16% slower, etc. The figures show that the overheads of our schemes are much lower than the software-based MPK virtualization *libmpk* [39]. The software-based MPK virtualization has almost the same

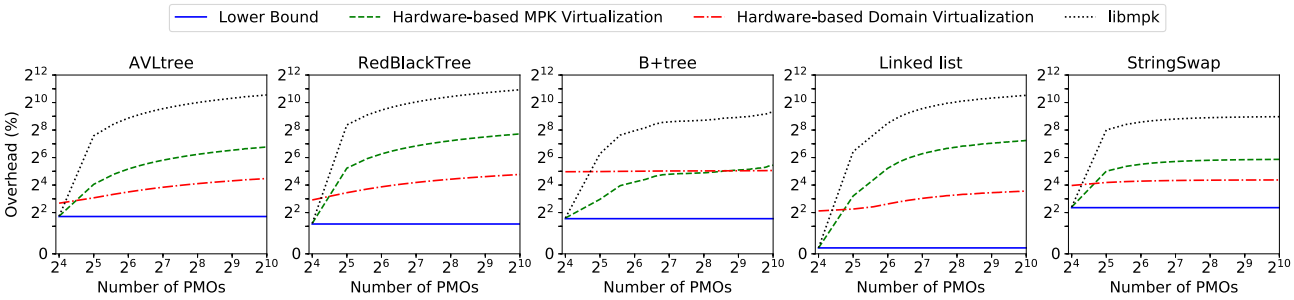

Fig. 6. Execution time overheads for the multi-PMO benchmarks as the number of PMOs varies, expressed as percentage slowdown over lowerbound, e.g. 2^2 means 4% slower, 2^4 means 16% slower, etc.

number of evictions as the hardware-based MPK virtualization design. Both of them need TLB invalidations after evictions. However, the software-based method needs to invoke system calls, `pkey_mprotect()`, to clean and set PTE bits in the page table. Our proposed MPK virtualization does not, and is hence several times faster.

Our domain virtualization design, in addition to the benefit of avoiding system calls, further removes the TLB invalidation requirement after every eviction. When the number of PMOs is small, hardware MPK virtualization provides a better performance than the Domain Virtualization method does, because the small eviction rate leads to few TLB invalidations, while the Domain Virtualization needs to look up PTLB for every domain access. When the number of PMOs is large, the advantage of the Domain Virtualization method becomes more obvious; its overhead is much less sensitive to the number of PMOs.

The location of the crossing point between the performance curves of the two hardware-based virtualizations depends on the data locality of the base application. On programs with a better locality, the crossing point tends to happen later (as PMOs increase). It is because on those programs, the TLB miss rate of accesses to PMOs is lower, while hardware-based MPK virtualization does not affect the performance of a TLB hit. For example, B+tree is a flatter tree (126 consecutive values in a PMO) than AVL tree and RedBlack tree, and hence it has a better data locality. It has a relatively smaller eviction rate, and a later crossing point in Figure 6.

On benchmarks with better data locality, all three methods show flatter curves in Figure 6 and lower overhead percentages. These programs have smaller buffer (DTTLB or PTLB) miss rates, hence lower eviction rates. The main overheads of the three methods are caused by evictions. An example is the comparisons between String swap and Linked list. The former has a better locality as for each swap operation, two 64-byte strings get swapped. There are 128 loads/stores incurring only up to two TLB misses. For the Linked list benchmark, each node access could cause a TLB miss, hence less flat curves.

Figure 7 shows the average of overheads of the five benchmarks in one figure. With 64 PMOs, the hardware-based MPK virtualization is $10.1\times$ faster than libmpk, while Domain Virtualization is $25.8\times$ faster. With 1024 PMOs, the hardware-based MPK virtualization is $10.6\times$ faster than libmpk, while the Domain Virtualization is $52.5\times$ faster than libmpk.

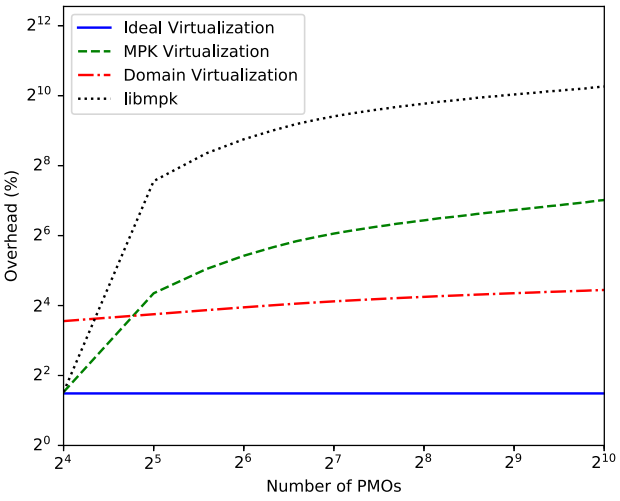

Fig. 7. Overhead comparison to libmpk [39] and lowerbound.

TABLE VII
OVERHEAD BREAKDOWN FOR THE PROPOSED SOLUTIONS WITH 1024 PMOS PER BENCHMARK.

Overhead sources	AVL	RBT	BT	LL	SS	Avg
Overhead of Hardware-based MPK Virtualization						
Permission change (%)	3.28	2.25	2.94	0.43	5.12	2.80
Entry changes (%)	0.05	0.08	0.11	0.01	0.18	0.09
DTT misses (%)	21.12	23.31	3.12	9.72	7.13	12.88
TLB invalidations (%)	84.56	180.6	37.44	143.4	48.05	98.81
Total (%)	109	206.3	43.61	153.6	60.48	114.58
Overhead of Hardware-based Domain Virtualization						
Permission change (%)	3.28	2.25	2.94	0.43	5.12	2.80
Entry changes (%)	0.04	0.07	0.09	0.01	0.16	0.07
PTLB misses (%)	15.83	18.68	2.02	7.52	5.04	9.82
Access latency (%)	3.09	5.23	28.27	4.34	15.49	11.28
Total (%)	22.24	26.23	33.32	12.30	25.81	23.97

Table VII reports the execution time overheads assuming 1024 PMOs used by each benchmark, broken down into sources of the overheads, for both our proposed schemes.

Hardware-based MPK virtualization suffers from larger overheads than Domain Virtualization, in particular primarily due to TLB invalidation overheads (contributing 98.81% of the total 114.58%). The average overheads of Domain Virtualization average 11.28% (ranging from 12.3–33.32%). Compared to Hardware-based MPK virtualization, Domain Virtualization removes TLB invalidation overheads but introduces PTLB access latency into the critical path. Since the former dominates execution time overheads, the trade off works very well. It is worth noting that the experimental setting is an extreme case where every access to PMO needs to switch the permission. If multiple accesses that are clustered together are protected by one pair of permission switches, the overheads would be lower.

C. Area Overheads

Table VIII reports the area overhead summary of the two designs. In hardware-based MPK virtualization, the DTTLB has 16 entries $\times$ 76 bits = 152 bytes. Assuming 1024 domains and up to 1024 threads per process, DTT is 256KB in size. There is one 64-bit register per process for the domain translation table to perform page table walk. TLB and PKRU are unchanged from MPK. In domain virtualization, PTLB is 16 entries $\times$ 2 bits = 24 bytes. DRT and PT are larger (16KB and 256KB) but are software data structures. Two 64-bits registers are added for DRT and PT.

TABLE VIII
AREA OVERHEAD SUMMARY OF TWO DESIGNS

	Hardware-based MPK Virtualization	Domain Virtualization
New Parts	1 64-bit register per core	2 64-bit registers per cores
	16 entries$\times$76 bits=152 Bytes buffer per core.	16 entries$\times$12 bits=24 Bytes buffer per core.
Other Changes	No	Extend 6 bits to each TLB entry (10% more)
Memory Usages	256KB memory per process per DTT	256KB + 16KB memory per process for DRT and PT

DTT, DRT and PT are software data structures, cacheable, and are placed in the paging system so they can be swapped in/out. Only DTTLB and PTLB require dedicated hardware tables and their sizes are negligible (both less than 0.2KB).

D. Security Analysis

For isolation between threads, both designs provide different memory views for each thread. The co-located attacker thread must insert special instruction, `SETPERM`, to change the permission to the target PMO for this thread. Then the attacker thread can read/write data to the target PMO. The insertion requirement simplifies the check on whether the injected code is malicious to PMOs. Code without `SETPERM` instruction cannot read/write data to PMOs. Attacks in code with `SETPERM` instructions and code trying to reuse `SETPERM` instruction in victim programs can be prevented by implementing call gates and performing binary inspection and rewriting similar to ERIM [50]. The isolation between threads is enforced.

For isolation within a thread, we insert `SETPERM` before and after every PMO accesses or every PMO data structure insert/delete operation, which gives a small window for memory access to a PMO within enabling and disabling pairs. `SETPERM` is implemented such that it is always followed by a memory fence. All memory accesses are finished before `SETPERM`, the instruction that enables permission to a PMO. So earlier memory accesses cannot read/write this PMO. Another `SETPERM` instruction disables permission to a PMO. Temporal memory safety is enforced within a thread. Attackers may exploit vulnerabilities in the code in the middle of a pair of the enabling and disabling instructions. However, the vulnerabilities and attacks are limited to this PMO and other enabled PMOs rather than all PMOs. For each thread, programmers can specify pair-wise interactions between PMOs. Any time, at most two PMOs are enabled. So the vulnerabilities and attacks are limited to at most two PMOs.

VII. RELATED WORK

MERR [60] proposed attach and detach primitives for PMOs to improve security between processes, which we build on in this paper. Another branch of papers enable more efficient memory encryption for persistent memory [5], [6], [13], [61]. There is a rich set of papers in literature covering other aspects of persistent memory, including but not limited to, memory-mapped files [14], [52], file system [15], [17], [58], [59], physical organization [3], [4], persistency models [2], [15], [28], [40], [46], [49], [52], logging [45], checkpointing [18] and GPU [33].

Software-fault isolation techniques (SFI) [43], [53] create a separate protected memory region by instrumentation at every memory access instruction. This ensures that the instrumented instruction can only access the designated memory segment; SFI incurs large overhead. ISboxing [16] separates address space to allow untrusted code to access only a 32-bit address space. The available address space reduction could limit practical usage of NVM. Jang et al. [25] propose to provide a heterogeneous isolated execution.

In the hardware aspect, the hardware approaches based on hardware page protection [7], [8], [12], [32], [35] support memory isolation and provide near zero overhead within a component. But switching between components still needs to switch to kernel mode, which incur substantial overhead. Frassetto *et al*. [19] try to provide in-process memory isolation which incurs high overhead to support many isolated domains. CHERI's [56] security hinges upon recompiling all external libraries rather than the application itself. However, our scheme works even for vulnerable libraries by limiting accesses (vulnerabilities) of libraries to the application. CODOM [51] needs dramatic changes to the hardware.

Hodor [20] provides isolated user-space libraries using MPK to improve the throughput and latency. Burow *et. al* [9] survey several shadow stacks implementations and propose a shadow stack implementation with MPK to reduce the shadow stack overhead.

ERIM implements call gates and binary rewriting and inspection to mitigate `WRPKRU` reusing by the attacker. libmpk [39] presents a software virtualization of MPK to support more domains. It incurs large overhead as the previous section has shown.

VIII. Conclusion

This paper proposes two architecture solutions to enable efficient intra-process isolation to enable domain-based PMO protection. The first solution, hardware-based MPK virtualization, augments MPK to remove the limit on the number of domains. The second solution, hardware-based domain virtualization, foregoes MPK and leverages some newly introduced mechanisms to avoid the large TLB invalidation overhead of the first method. The experiments show that the two solutions provide much reduced runtime overhead compared to the previous solutions. Even in the extreme case (permission switches for every PMO access), the Domain Virtualization method is subject to less than 24%, over $50\times$ reduction of the overhead of a previous state-of-the-art software solution.

Acknowledgement

We thank all the anonymous reviewers whose feedback is helpful for improving the final version of the paper. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1900724, CCF-1525609, CNS-1717425, CCF-1703487, and Office of Naval Research (ONR) under grant No. N00014-20-1-2750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

References

[1] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[2] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy Persistency: a High-Performing and Write-Efficient Software Persistency Technique," in *Proc. of the International Symposium on Computer Architecture*, 2018.

[3] A. Awad, S. Blagodurov, and Y. Solihin, "Non-Volatile Memory Host Controller Interface Performance Analysis in High-Performance I/O Systems," in *Proc. of the International Symposium on Performance Analysis of Systems and Software*, 2015.

[4] A. Awad, S. Blagodurov, and Y. Solihin, "Write-Aware Management of NVM-based Memory Extensions," in *Proc. of the International Conference on Supercomputing*, 2016.

[5] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers," in *Proc. of the International Symposium on Architecture Support for Programming Language and Operating Systems*, 2016.

[6] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "ObfusMem: a Low-Overhead Access Obfuscation for Trusted Memories," in *Proc. of the International Symposium on Computer Architecture*, 2017.

[7] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged {CPU} features," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 335–348.

[8] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments." USENIX Association, 2008.

[9] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 985–999.

[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 52:1–52:12.

[11] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu, "Efficient support of position independence on non-volatile memory," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 191–203.

[12] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 56–71.

[13] S. Chhabra and Y. Solihin, "i-NVMM: A Secure Non-Volatile Main Memory System with Incremental Encryption," in *Proc. of the International Symposium on Computer Architecture*, 2011.

[14] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 47, no. 4, pp. 105–118, 2012.

[15] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 133–146.

[16] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 555–566.

[17] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[18] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2017.

[19] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "{IMIX}: In-process memory isolation extension," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 83–97.

[20] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019.

[21] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[22] Intel, "Intel 64 and ia-32 architectures software developer's manual." https://software.intel.com/en-us/articles/intel-sdm, online; accessed 11 November, 2019.

[23] A. R. Intel, "Persistent memory programming," http://pmem.io/.

[24] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[25] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 455–468.

[26] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 660–671.

[27] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro *et al.*, "2mb spin-transfer torque ram (spram) with bit-by-bit bidirectional current write and parallelizing-direction current read," in *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. IEEE, 2007, pp. 480–617.

[28] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 399–411, 2016.

[29] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 437–452.

[30] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 256–267.

[31] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, pp. 143–143, 2010.

[32] H. Lee, C. Song, and B. B. Kang, "Lord of the x86 rings: A portable user mode privilege separation architecture on x86," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1441–1454.

[33] Z. Lin, M. Alshboul, Y. Solihin, and H. Zhou, "Exploring memory persistency models for gpus," in *Proc of International Conference on Parallel Architectures and Compilation Techniques*, 2019.

[34] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 411–425.

[35] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1607–1619.

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[37] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1. ACM, 2017, pp. 135–148.

[38] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[39] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel {MPK})," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 241–254.

[40] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 265–276.

[41] Riku, Antti, Matti, and N. Mehta, "The heartbleed bug," *http://heartbleed.com*, 2014.

[42] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[43] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," 2010.

[44] H. Shacham *et al.*, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." in *ACM conference on Computer and communications security*. New York,, 2007, pp. 552–561.

[45] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 178–190.

[46] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 175–186.

[47] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 574–588.

[48] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 2015.

[49] Y. Solihin, "Persistent memory: Abstractions, abstractions, and abstractions," *IEEE Micro*, vol. 39, no. 1, pp. 65–66, 2019.

[50] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "{ERIM}: Secure, efficient in-process isolation with protection keys ({MPK})," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1221–1238.

[51] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "Codoms: Protecting software with code-centric memory domains," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 469–480.

[52] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[53] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 203–216.

[54] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, "Hardware supported persistent object address translation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 800–812.

[55] T. Wang, S. Sambasivam, and J. Tuck, "Hardware supported permission checks on persistent objects for performance and programmability," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 466–478.

[56] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[57] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 457–468.

[58] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[59] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. D. Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[60] Y. Xu, Y. Solihin, and X. Shen, "Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 987–1000.

[61] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 479–492.